VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MRI DATA PROCESSING ACCELERATION ON GPU

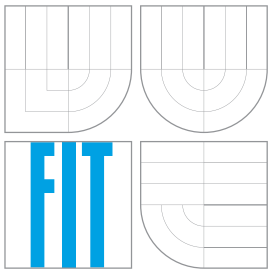
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP KEŠNER

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AKCELERACE ZPRACOVÁNÍ DAT Z MRI NA GPU

MRI DATA PROCESSING ACCELERATION ON GPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — FILIP KEŠNER
AUTHOR

VEDOUCÍ PRÁCE — prof. Dr. Rolf Krause , Ing. Lukáš Polok
SUPERVISOR

BRNO 2012

Abstrakt

Identifikace trajektorií neuronových vláken uvnitř lidského mozku má velký význam v mnoha lékařských aplikacích, jako neurologická diagnostika, neuro-navigace, léčba epilepsie, chirurgické operace a tak dále. Za použití dat z MRI, metod postavených na Markovských řetězích a Monte Carlu mohou být možné trajektorie vypočítany a ty nejpravděpodobnější zobrazeny. Tyto informace o trajektoriích mohou sloužit jako vstup pro pokročilé metody lékařské diagnotiky a léčby. Vzhledem k obrovskému množství dat a velkého počtu iterací toto může být časově náročný proces. Za účely, jako jsou statistická analýza a/nebo porovnávání několika datových sad a/nebo pacientů, požadavky na výpočetní čas jsou enormní. Rychlejší diagnóza může také přinést nasazení léčby dříve. Nyní existuje jen velmi málo implementací softwaru pro neurální traktografii. Implementací softwaru pro pravděpodobnostní neurální traktografii je ještě méně. Nynější implementace, provádějící všechny operace postupně na CPU, jsou značně pomalé. Účelem této práce je poskytnout efektivní implementaci, která vvyužíva GPU. Za účelem implementace na GPU, je poskytnuto porovnaní technologií CUDA a OpenCL.

Abstract

The identification of trajectories of neuron fibres within the human brain is of great importance in many medical applications as the neural diagnostics, neuronavigation, treatment of epilepsy, surgical removal of tumors and etc. By using diffusion MRI-data as input, and by employing Monte-Carlo like methods, possible trajectories are generated and the most likely ones can be visualized. These can serve as input for advanced medical diagnosis and treatments. Due to the huge amount of data to be analyzed and many iterations, this is a time consuming process. For the purposes such as statistical analysis and comparsion over several datasets or several patients, computational time requirements are enourmous. Faster diagnosis can improve routine throughput and provide earlier treatment of illness. At this time there exists only a very few implementations of neural tractography sof tware. For probabilistic neural tractography is the list of software even thiner. Today's implementations using standard serial CPU execution suffer from high time consumption. The goal is to provide an efficient implementation which makes use of GPGPUs and exploits parallelism in the method. For the GPU implementation, a comparsion of CUDA and OpenCL technologies will be provided, using the more suitable one.

Klíčová slova

neurální traktografie, pravděpodobnostní traktografie, nervová vlákna, magnetická rezonance, GPU, GPGPU, CUDA, OpenCL, CPU, paralelní výpočty, akcelerace výpočtů

Keywords

neural tractography, probabilistic tractography, neural fibers, MRI, DTI, GPU, GPGPU, CUDA, OpenCL, CPU, parallel computing, computational acceleration

Citace

Filip Kešner: MRI Data Processing Acceleration on GPU, bakalářská práce, Brno, FIT VUT v Brně, 2012

MRI Data Processing Acceleration on GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením panů prof. Dr. Rolfa Krause a Ing. Lukáše Poloka.

.....
Filip Kešner
July 31, 2012

Contents

Acknowledgements

I would like to thank my supervisor Prof. Dr. Rolf Krause, who was continuously supporting me in solving problems, that I met during the work on this report. I especially thank him for his encouragement and his guidance. Writing this thesis have been for me very valuable experience especially thanks to him.

I would like also to thanks Valentina Poletti for great language corrections, and to Dorian Krause for helping me to solve nVidia GPU related issues that I encounter during my experiments and testing. Thanks also belong to my local advisor Ing. Lukas Polok for his advices.

My thanks also belongs to my friend, and colleague,
neural research specialist Dr. Jiri Keller
from the Charles University, Prague.

Filip Kesner

Chapter 1

Introduction

This document's purpose is to describe the approach to the acceleration of the probabilistic neural tractography processing, also called neural trajectories identification.

1.1 Motivation

The motivation for the neural tractography is primarily to provide the new way to study and to make a diagnosis of human brain. This advanced modern technique can bring us the very specific knowledge about subject's brain structure and also about the possible damage or disorders.

At this time there exists only a very few implementations of the neural tractography software. For the probabilistic neural tractography is the list of software even thiner. Today's implementations using the standard serial CPU execution suffer from the high time consumption. This problem is slowing down the neuro research and neuro diagnostic teams around the world.

The neural tractography is in more detail described in the following chapter.

1.2 Goal

The probabilistic neural tractography can be very computationally slow. The goal of this work is to provide an efficient implementation, which makes use of GPGPUs and exploits a parallelism in the method.

To perform this task several steps were followed:

- understand how the neural tractography works
- analyze the existing implementation
- identify possibilities for speed up
- select the proper GPU technology
- use benefits of the GPGPU to perform tasks in the parallel manner

We will dedicate the next part of the text to introducing the neural tractography and approaches used to solve this task.

Chapter 2

Problem description

2.1 Neural Tractography

Neural tractography is a modern (very recent) medical analysis approach dedicated to providing knowledge on neural fiber trajectories primarily inside the brain of a subject.

To perform this very complex task, special technique of magnetic resonance imaging (MRI) is used. Recorded data then goes through computer based post-processing (algorithms).

Neural tractography can provide information about long fiber trajectories which interconnect brain with the rest of the body but information gained using this analysis can also provide knowledge of interconnection inside very complicated 3-dimensional network formed by short connections among various cortical and subcortical regions of the human brain.

Results of this process have to be readable for medical neuro-specialists and neurosurgeons, the visualization is presented as 2-dimensional or 3-dimensional image, usually using colormap instead of monochromatic image otherwise typical for MRI results.

Brain neural tracts were impossible to identify by direct examination (invasive method), computed tomography (CT) or basic magnetic resonance imaging (MRI) scans until special magnetic resonance imaging method called diffusion tensor imaging (DTI) was developed.

Neural tracts can be imagined as bundles of neural fibers sharing the same trajectory, usually also direction. But this analysis method cannot provide information if neural fiber is for example heading from left part of brain to right one or vice versa, what we get is knowledge that there exists neural connection. In fact lack of this information is not critical, lets imagine an ethernet computer network, typically all communication links are both-directional so if the link is disrupted, communication connection is down for both directions.

2.1.1 Computed tomography (CT)

Computed tomography is a radiological diagnostic method, which uses X-ray emissions from several directions around the subject. Then acquired images are processed on the computer, which allows to visualize internal tissues of the usually human subject.

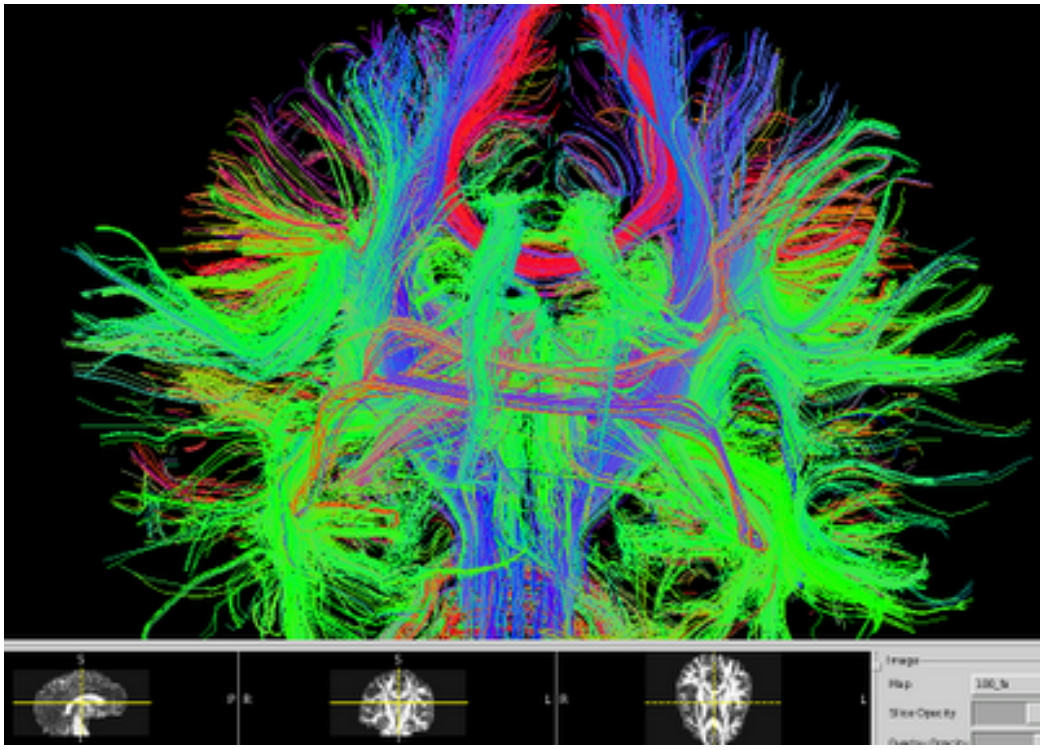


Figure 2.1: Tractography color-map visualization

2.1.2 Magnetic resonance imaging (MRI)

Magnetic resonance imaging (MRI) is medical imaging technique which allows us to inspect internal parts of body, including brain, without invasive surgery. Area of medicine which uses MRI as one of the diagnosis techniques is called radiology.

Magnetic resonance provides sufficient contrast to distinguish different soft tissues of the human body. This is very useful in imaging (visualising) the brain, heart, muscles, cancer, etc.

In comparison with computed tomography (CT) which uses X-rays, MR uses strong magnetic fields and does not use ionizing radiation which makes it less harmful to a human body and relatively safe.

The principle by which magnetic resonance works is based on the knowledge that a human body contains a huge amount of water (around 70-90% of body weight). Water represents in physical terms molecule containing 2 atoms of hydrogen and 1 atom of oxygen. For basic MRI strong directional magnetic field is used that aligns hydrogen nuclei along the direction of the field. After that, a radio transmitter creates electromagnetic field at the right frequency, also known as the resonance frequency. This electromagnetic energy is absorbed and forces the protons spin flip in magnetic field. Additional electromagnetic field is then switched off, allowing spins of the protons to return into their previous thermodynamic equilibrium. During the process of re-aligning along the strong magnetic field, nuclei are loosing energy received by the electromagnetic transmitter, this energy is radiated as

electromagnetic waves, which are sensed by receiver coils.

After this process is carried out, an image is constructed based on results of the inverse Fourier transformation. This is done thanks to a fact that the protons in different tissues take different times to get realigned with the strong magnetic field which also provides a phase delay.

This was a simple introduction to a basic magnetic resonance principle of function, for more details you can use [20].

2.1.3 Diffusion tensor imaging(DTI)

DTI [2] is magnetic resonance technique specifically developed to provide more detail information than basic magnetic resonance imaging.

DTI uses readings of diffusion of water in the body tissues, which can reveal its 3-dimensional shape.

Free diffusion of water in the body tissue exists equally in all directions, this free diffusion is called „isotropic“. This can be thought of for better understanding as a molecule movement inside a sphere.

In the case where water inside the tissue is closed inside barriers, the diffusion occurs more likely in direction which is not blocked by barriers, this directional diffusion is called „anisotropic“. Water can be blocked by many barriers like: cell membranes, myelin, axons, ... For neural tractography shape of neural axon is very interesting, as we can see in the image, it can provide directional information[3].

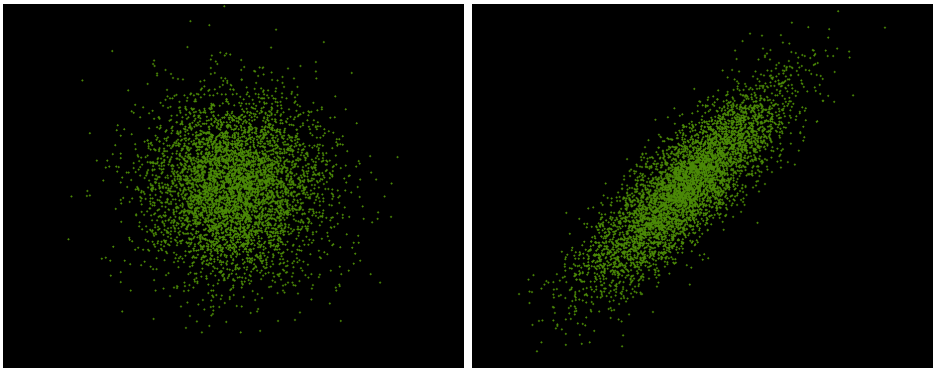


Figure 2.2: Isotropic and Anisotropic diffusion

Of course this technology precision is unable to detect a single neuron, which would not provide any big benefit, for trajectory identification is important indicator of the order of neural axons, in other words: bundle of neurons, which axons are heading same direction is sensed as increased anisotropy inside a voxel. [1]

[1] generic information about voxel can be found in [26]

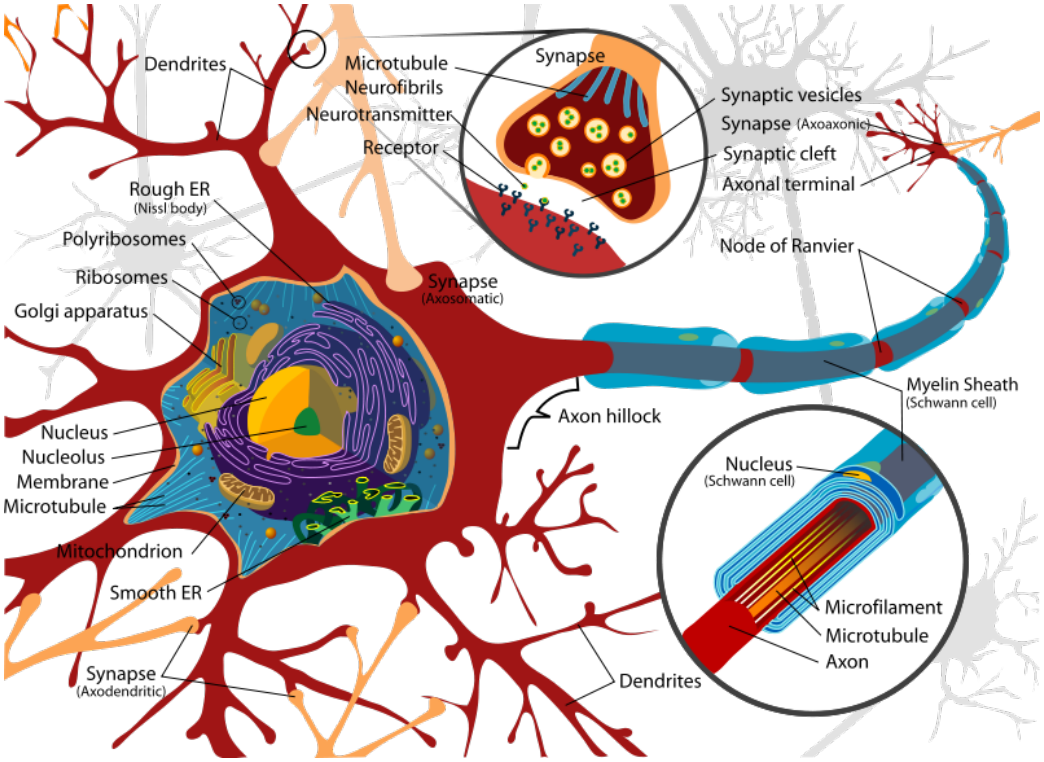


Figure 2.3: Neuron illustration [19]

At this state of technology the analysis based on the recorded voxel data is used with the minimal voxel size 2x2x2 mm at clinical 1.5T scanners and less then 1.7x1.7x1.7 mm at 3T.

A voxel, also called volumetric pixel or volumetric picture element, represents a value in 3-dimensional space. This is analogous to a pixel, which represents a value in 2-dimensional image. We can imagine a voxel as 3-dimensional cuboid which is the smallest readable element in the application.

Information purely based on the intensity of anisotropy should be analysed with the knowledge that anisotropy can be alternated also by other influences also. For example: conditions of myelin or axon structure disruption such as trauma, tumor or inflammation reduce anisotropy, because barriers are partially destroyed or disorganized.

Anisotropy is measured in several ways. One way is by a ratio called fractional anisotropy (FA). An anisotropy of „0“ corresponds to a perfect sphere, where „1“ is an ideal linear diffusion. Based on observations, well-defined tracts have FA higher than 0.20 but few regions have fractional anisotropy higher than 0.90. [24] This number provides information about how aspherical the diffusion is, but says nothing about the direction.

More information about diffusion MRI can be found int [12].

Chapter 3

Methods description

In this chapter a description of methods, used in probabilistic neural tractography, will be provided.

To make neural tractography possible, we need a specific type of magnetic resonance imaging method called „diffusion weighted imaging“[4], which was briefly described in the previous chapter. Data provided by DTI can be interpreted in several ways, using different models of diffusion within voxel:

- Diffusion tensor model
- Simple partial volume model

3.1 Diffusion tensor model

The diffusion tensor model is often used to describe diffusion inside a voxel as a 3-dimensional tensor using a covariance matrix to compute gradients, assuming directions of diffusion along the gradients.

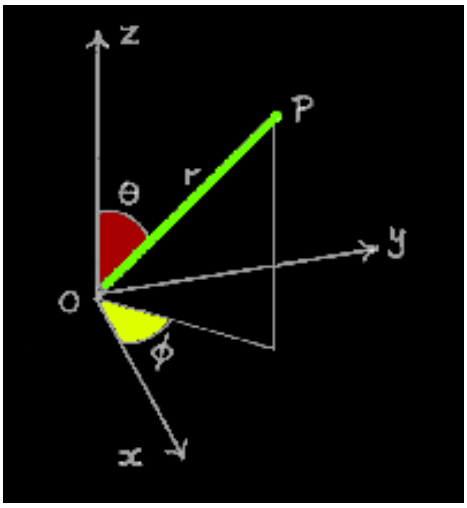


Figure 3.1: Spherical polar coordinates

3.2 Simple partial volume model

This simple partial volume model is slightly different from the diffusion tensor model. The important difference is that, instead of modeling the diffusion shape (ellipsoid) directly using MR data, this approach is based on modeling underlying fiber structure, using prediction of the diffusion shape and fitting MR data in.

The simplest partial volume model can be built on an assumption, that all fibers within the voxel share the same orientation. This approach assumes no diffusion-diffusion exchange, which results in simple two compartment partial volume model.

The first one models diffusion inside and around axons using only fiber direction diffusion. The second one models diffusion of free water, this represents isotropic diffusion.

The main difference resulting from this modeling approach is that the diffusivity in all directions, which are perpendicular to fiber axis, are assumed(constrained) to be equal.

If we would use simple partial volume model as presented above, it would result in only one fiber orientation modeled per voxel. This simplification would result in loss of information. To prevent loss of information we can divide each voxel in the several subvoxels and use simple partial volume model for them. Now we assume that the MR signal recorded for each voxel is a sum of the signal from „virtual“ subvoxels and within each subvoxel only one fiber direction is modeled.

Using this assumptions, signal from subvoxel can be described with following formula:

$$\mu_i = S_0((1-f)\exp(-b_i d) + f\exp(-b_i d \vec{r_i}^T \vec{R}\vec{A}\vec{R}^T\vec{r_i}))$$

When a diffusion profile of this signal is measured, we are measuring a variant of this signal which is smoothed in angular space, with a kernel, predicted by this model, of $exp((-bdcos^2(\gamma))$.

$$\mu_i = \sum_{(\theta,\phi)\in\Theta\Phi} \left(\sum_{j\in V_{\theta\phi}} \frac{S_{0_j}}{N} \left[(1-f_j)\exp\left(-b_i d_j\right) + f_j \exp\left(-b_i d_j r_i^T \vec{R}_{\theta\phi} A \vec{R}_{\theta\phi}^T r_i\right) \right] \right)$$

The important information of interest to us is voxel fiber orientation probability distribution inside each voxel. This is also called local parameter estimation, because it is based on local diffusion. To compute local parameter estimation Markov chain Monte Carlo method is used. Markov chain Monte Carlo will be described further in the text.

3.3 Monte Carlo method

In fact Monte Carlo is not one method, it is a group of methods / computational algorithms which use random sampling. Monte Carlo methods are often used in computer simulations for physics computations, computational biology, mathematical modeling, etc. Especially this group of methods is very useful in situations, where classical deterministic precise computation approach would be infeasible.

To give example of such situation, related to neural tractography take a look on input data for computing primary (dominant) diffusion tensor orientation. As DWI MRI scan is performed [4], MRI device uses different angles and different intensities to visualize the water diffusion. So as the result of this process we have a data acquired in form of the list containing: directions, scan intensity and diffusivity per each voxel (for each direction and intensity of scan).

3.4 Markov chain

In brief description, the Markov chain is a mathematical system, which is based on transitions from one state to another, using a finite number of states. With some simplifications we can imagine the Markov chain as a non-deterministic finite state machine using probabilities to describe transition rules. For Markov chain is characteristic its absence of memory, which means that next possible state is determined only by current state and probabilistic transitions. This feature is called „Markov property“.

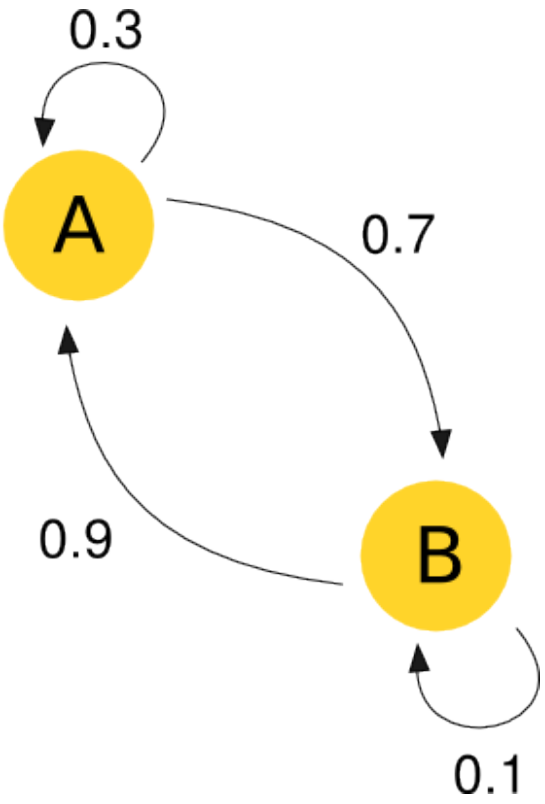


Figure 3.2: Markov chain diagram

Markov chain is very often used in statistical models of real world, in our context we are using Markov chain in combination with Monte Carlo methods.

Markov chain is described by using 2 mathematical structures:

vector of absolute probabilities
$p(n) = [p_1(n), p_2(n), p_3(n)..p_i(n)]$

where $p_i(n)$ describes probability that at the moment n system is in the state i

probability matrix of transitions
$P(n) = [p_{i,j}(n)]$, where $i = 1, 2, .. N$ and $j = 1, 2, .. N$

Probability $p_{i,j}$ needs to comply to following constraint
$p_{i,j} >= 0$

and a sum of probabilities in each line of the matrix needs to be equal to 1, because it is an absolute system of events
Markov chain can be also described as:

$$p(n+1) = p(n) * P$$

and using sequential substitutions we can reach following:

$$p(n+1) = p(0) * P^{n+1}$$

3.5 Markov chain Monte Carlo

Markov chain Monte Carlo, also called by shortcut MCMC, is a group of methods that uses Monte Carlo random walks in order to sample from probabilistic distributions provided by Markov chain with the desired probability distribution [1].

For solving integrals, unsolvable by analytical methods, is possible to use several computation methods. We can plot samples in the space using all parameters as dimensions and than perform random walks by randomly choosing parameters values. Then for an evaluation of results (rejecting or accepting based on criteria) we can use different methods. Lets mention some of them. The first possibility is *rejection sampling*, the second one is *importance sampling*, of course this is not a complete list of possibilities. This approach can provide true random sampling which can generate independent samples, but also thanks to true random sampling only a minority of performed samples would be accepted. This property would result in low computational efficiency, especially in high dimensional parameter spaces.

MCMC is a sampling technique which in our context addresses the problem by proposing samples preferentially in areas of high probability. Samples in this case are no longer independent of each other, but the highly increased probability of accepting the sample is ,in this otherwise computationally expensive case, the more important benefit.

Construction of the Markov chain with the desired properties is usually not difficult, but determining (estimating) the required number of steps to achieve the stationary distribution

with acceptable error can be. Typical use of MCMC can only provide an approximation of the target distribution, we should not forget that there will be every time the effect of starting position.

When we have chosen one local model also known as model of diffusion we can proceed to making an estimation of neural interconnection trajectories.

3.6 Global interconnection estimation

The goal of neural tractography is to make as much precise the estimation of neural fibers trajectories as possible.

As we saw earlier in this chapter, there exist several approaches to model local probabilistic diffusion within a voxel and it is very important to choose the appropriate local diffusion model to make an effective and reasonable match with the global connectivity model (groups of interconnected voxels). To achieve probability results of each possible neural fiber trajectory we will use those local probability density functions (pdf) gathered by using simple partial volume model for the local diffusion description.

Motivation for using the probabilistic approach which provides not only trajectories but also their probabilities instead of just finding the most probable ones is that we will know better how much we can rely on that information. Another benefit of this probability results providing approach is, that complex fiber structures will be represented by the much higher uncertainty in principal direction instead of just one most probable trajectory, which would not provide any use and could even create confusion in medical analysis.

If we would use local diffusion parameters without any uncertainty telling us only one direction of fiber, we would be able only to find the most obvious trajectories, this model could provide only limited information.

Using this simple binary approach we can „walk trough“ most obvious trajectories from starting position A using a seed algorithm and continuing in the direction provided by the diffusion tensor.
This algorithm need to be run for each voxel, which makes it computationally intensive.

3.6.1 Interpolation

The sampling technique used in this case requests local probability distribution functions (pdfs) existing in continuous space. Data acquired from the MR scan, preprocessed by computing local parameter estimations, lay on a discrete grid. To transform these discrete grid data into continuous space we would need a mathematical technique which is called an *interpolation*. Interpolation technique can provide estimation of values in the area of space between grid points of the measured data. For this purpose we could use one of the several interpolation techniques such as **sinc** or **trilinear** interpolation, but it would be very computationally expensive and it could also amplify the noise influence in complex neural structures.

Based on this knowledge, a different approach is chosen. This alternative approach uses the samples on grid directly from one of the neighbor voxels. In a probabilistic system the possibility to use a probabilistic interpolation is also provided. This could be described as selecting neighbor voxel on grid, but probabilities of selecting each neighbor will be a function g. There exists many functions for g, but in this case one analogous to trilinear interpolation was selected.
More detail description of this interpolation can be found in [8]

3.6.2 Stopping conditions

Streamline generating algorithms based on maximal probability of fiber direction (provided by diffusion tensor in voxels) need some defined streamline conditions or criteria to distinguish different fibers from each other and to distinguish neural fibers and complex neural structures from each other. These stopping conditions are usually based on fractional anisotropy and local curvature (angle difference between actual and following steps). Fractional anisotropy thresholds has been estimated in the range of 0.2 - 0.4 by empirical experience, more about this can be found in referenced document [5]. Curvature, angle difference thresholds have been set to strict condition, that the angle difference has to be in range of -45 to +45 degrees. These criteria settings are needed to reduce the influence of the noise in the image, partial volume effects, etc. and most importantly to prevent seeing false positives in the results by only making progress (next step in the streamline algorithm) when there is high confidence in fiber direction and when the direction is anatomically plausible. The drawbacks of this strict criteria approach are limitations strictly imposing on which fiber tracts may be reconstructed and also where in the brain they are plausible.

To make this a little more clear, we will provide a small example:

Deep gray matter tends to have low anisotropy, which is usually bellow the threshold for streamlining algorithms mentioned earlier. Streamlines in this case also do not progress into the cortex, because anisotropy is decreasing in these areas of brain which implies increasing uncertainty of fiber direction.

As we saw the drawbacks of the maximal probability streamlining approach. Lets look on the different approach based on the probabilistic algorithm. The probabilistic algorithm has several significant advantages in comparison with the last mentioned one.
The advantages are following:

In the areas of brain where fiber direction is uncertain (usually the same regions where anisotropy is low) probabilistic algorithm can provide a direct representation of that uncertainty. Even in the situations where the probabilistic streamline algorithm cannot progress along a singe direction with the high level of confidence, a progress can be made in many directions. The level of uncertainty in these areas is then represented by voxels further along the path, marked by lower probabilities, but still the high probability of fiber connection to the seed (originating) voxel from actual one with higher uncertainty is associated with the area where the path progresses.

Probabilistic algorithms have a high level of resistance against noise influence. This might be very beneficial in praxis, when tracking fibers near brain tumor, where the diffusion

properties can be heavily influenced by this pathology and where FA is typically decreased. In the situations, where the path progresses into a noisy voxel using non-probabilistic algorithm may encounter problematic situation, ending maybe with false change in the path. The probabilistic algorithm solves these situations naturally by decreasing the probability of paths which tends to quickly disperse wrong path, marking it with low probability, but the true paths with strong probabilities tend to group together, even increasing the probability.

Thanks to these advantages of probabilistic algorithms, the need of anisotropy and curvature stopping conditions is significantly reduced. The anisotropy constrains (thresholds) can be absolutely avoided. For the curvature some constraints should remain to prevent tracking back along a path similar to one already visited, which would cause inadequate increasing the probability along the path. To make sure that this will not happen, we check at very step if the path enters an area already visited, if this is true, path is terminated. The angle constraint was empirically estimated to range from -80 to +80 degrees.

Lets take a look at what is in a fact the result of using all these methods and algorithms together in this probabilistic tractography context.

3.6.3 Results

In this context we can think of this probability distribution as a distribution of connections from the seed point. But that would not be right in this case.
Based on the model which we described earlier in this chapter, these spatial pdfs represents confidence bounds on the location of the most probable single connection.

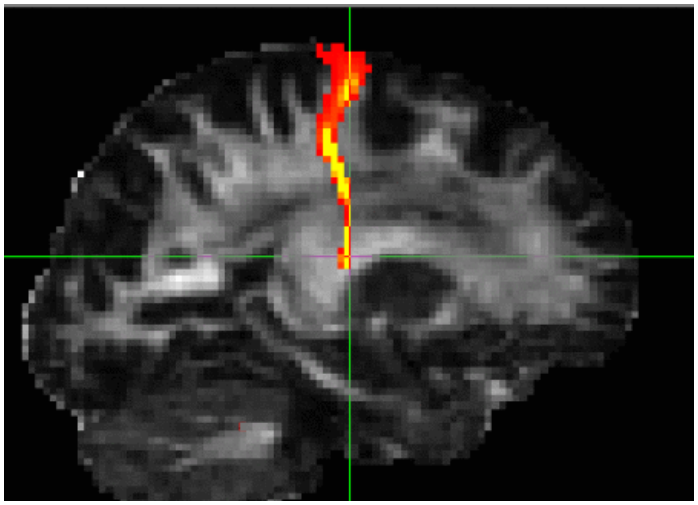


Figure 3.3: Neural tract probability connection from voxel of interest

Chapter 4

State of the art

The concept of „State of the Art“ is defined as, „The level of knowledge and development achieved in a technique, science“, used in term of the highest level of development in particular time. [23]

4.1 Software

In this subsection will be presented available software for neural tractography. The following software description is not describing absolutely all software related to neural tractography or Diffusion Tensor Imaging, but the most used software, by neuro research teams around globe, in this specific area.

In a short list we introduce the following software:

- FSL,
- MedINRIA,
- brainVoyager,
- TRACULA,
- Camino

All this software is supported on GNU/Linux and Microsoft Windows platforms. But not all of this software is an open source or even available for free.

4.1.1 FSL

FSL(FMRIB Software Library) is a comprehensive library of analysis tools for FMRI, MRI and DTI brain imaging data. FSL is written mainly by members of the Analysis Group, FMRIB, The Oxford Centre for Functional MRI of the Brain

According to FMRIB homepage FSL is in active use in over 1000 laboratories around the world.

FSL in generic http://www.fmrib.ox.ac.uk/fsl/

FSL is a very generic set of software for various types of MRI scans.
The related diffusion magnetic resonance imaging subset of software is:

FDT - FMRIB's Diffusion Toolbox - tools for low-level diffusion parameter reconstruction and probabilistic tractography, including crossing-fiber modelling.

TBSS - Tract-Based Spatial Statistics - voxel-wise analysis of multi-subject diffusion data.

FSL provides a subset of software dedicated to neural tractography based on DTI MRI [15]. This subset is called FDT (FMRIB Diffusion Toolbox). It contains several useful pieces of software. The most important and relevant ones for this thesis are listed below:

bedpostx - software dedicated to the local modelling of diffusion parameters, this very important and relevant software will be described later in the text

probtrackx - software for tractography and connectivity-based segmentation, this software uses the output of bedpostx (which with certain simplification can be viewed as diffusion direction per voxel map) to compute probabilistic fiber trajectories. This software will be also described in more detail later in the text

FSL FDT homepage: http://fsl.fmrib.ox.ac.uk/fsl/fdt/

FSL FDT is at this time one of a very few software packages, available and being widely used, which provides a probabilistic approach to neural tractography. In other words fiber trajectory estimation which results into not only trajectory but also in probability (reliability) of that trajectory.

4.1.2 MedINRIA

MedINRIA is a free collection of software developed within the Asclepios research project. It aims at providing clinicians with state-of-the-art algorithms dedicated to medical image processing and visualization.

http://www-sop.inria.fr/asclepios/software/MedINRIA/

DTI Track module of MedINRIA

The DTI Track module provides all necessary tools for in-deep DT-MRI analysis and fiber tracking. From diffusion tensor field estimation to the FA/ADC computation, across the tensor smoothing, and fiber extraction. This module helps to extract a fiber bundle of interest. Moreover, it is possible to fuse fMRI (functional MRI) data with fiber pathways, to determine likely paths linking activated regions. A brief descriptions of the features is given below.

The important parts of the DTI Track module of MedINRIA for this thesis are providing:

DTI Analysis:

Fast tensor estimation, tensor smoothing and DTI analysis. Scalar maps calculation: FA (Fractional Anisotropy), ADC (Apparent Diffusion Coefficient), etc.

Fiber Tracking:

Powerful fiber tracking algorithm, using tri-linear Log-Euclidean interpolation. The algorithm is fully multithreaded.

4.1.3 BrainVoyager QX

BrainVoyager QX is commercial application with closed source code. It claims to be a highly optimized software package for the analysis and visualization of functional and structural magnetic resonance imaging data sets.

For its closed source and commercial approach of developers, it seams to be an unfavourable software to be accelerated by external developers.

http://www.brainvoyager.com/products/brainvoyagerqx.html

4.1.4 TRACULA

TRACULA - TRActs Constrained by Under-Lying Anatomy
is a tool for automatic reconstruction of a set of major white-matter pathways from diffusion-weighted MR images. This tool is part of FreeSurfer related software.

It uses global probabilistic tractography with anatomical priors. Prior distributions on the neighboring anatomical structures of each pathway are derived from an atlas and combined with the FreeSurfer cortical parcellation and subcortical segmentation of the subject. This is being analyzed to constrain the tractography solutions. A benefit of this approach is the possibility to avoid the need for user interaction, for example: to draw ROIs (region of interest) manually or to set thresholds on path angle and length, and thus to automate the application of tractography to large datasets.

TRACULA is very new piece of software, according to its homepage, it was released in May of 2011. This software can have a bright future but at this time is almost unknown in neural research community.

http://surfer.nmr.mgh.harvard.edu/fswiki/Tracula

4.1.5 ULC Camino

UCL Camino Diffusion MRI Toolkit
Camino is an open source software. Its core is primarily implemented in Java. The toolkit implements standard techniques, such as diffusion tensor fitting, mapping fractional anisotropy and mean diffusivity, deterministic and probabilistic tractography. It also contains more specialized and cutting-edge techniques, such as Monte-Carlo diffusion simulation, multi-fiber techniques, compartment models, and axon density and diameter estimation.

This software also brings possibility to run probabilistic tractography. Community around this ULC research group is related to Derek Jones, an important person, in diffusion magnetic resonance imaging community.

http://cmic.cs.ucl.ac.uk/camino//index.php?n=Main.HomePage

4.1.6 Selected software base

Here we will discuss reasons for selecting the specific software as the base for the acceleration.
To make the acceleration of software possible, several requirements have to be met.

From the legal point of view, the software license has to be compatible with the alternation of the source code by external developers. This can almost imply that package should be under some kind of an open-source compatible license.

Also to provide an effective implementation, the base source code of software should be written in some computational effective language(C, C++, ...). Or at least, there have to be a possibility to link together with some computationally effective language.

If we take in the consideration the contribution of this work to others, users of neural tractography software, it would be wise to accelerate the most widely used software in this area. After discussion with several medical neuro specialists from Czech republic and United Kingdom, and including my independent search, the most preferred candidate was FSL FDT.

FSL FDT met all requirements mentioned above. This software was originally implemented in Oxford by FMRIB Analysis Group [1], The Oxford Centre for Functional MRI of the Brain .

FSL FDT (Diffusion Toolbox) contains several independent software packages which together provides all needed processing in neural diffusion tensor imaging.

[1] FMRIB homepage: http://www.fmrib.ox.ac.uk/

Here is the process of neural trajectories identification(tractography) divided into phases according [7] . Also illustration time consumption is mentioned using following data parameters:

```
Intel 2.4 GHz processor
60-direction whole brain dataset
dimensions 128 x 128 x 64
2.5 mm isotropic resolution
```

- MRI scanner-specific preprocessing - manually done by the user
- Eddy current correction - using FDT (around 3 minutes per volume).
- BET - brain area extraction
- DTIFIT - fitting of diffusion tensors to check data quality (1 minute)
- BEDPOSTX - fitting of the probabilistic diffusion model on data, diffusion mapping (20 hours)
- FDT_REG - registration (3 min)
- PROBTRACKX - probabilistic tractography, generating connectivity distribution, based on diffusion map (10 sec per voxel of interest)

As we can directly see, the diffusion mapping (fitting diffusion model) consumes the highest amount of computational time. Based on this knowledge, the acceleration of the BEDPOSTX became priority over other software packages within the FDT.

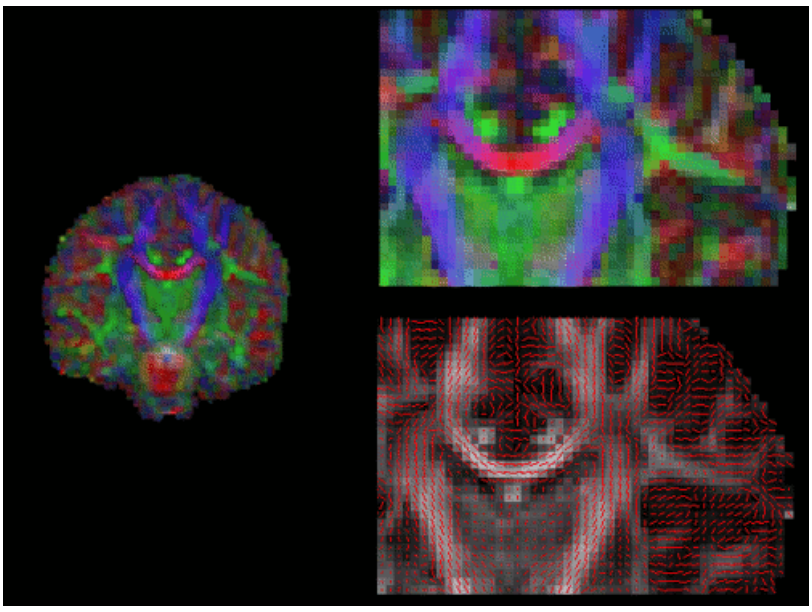


Figure 4.1: BEDPOSTX output: primary diffusion tensor orientation map

Chapter 5

General purpose GPU computing

5.1 GPGPU

In this chapter a relatively new approach is described, which is called GPGPU, General purpose GPU computing. Simply said, this computational approach uses graphical processing units to perform specific parallel computation to speed-up the program execution and to unburden central processing unit (CPU).

In this short introduction to GPGPU, it would be wise to introduce the concept of GPU first. GPU (Graphical Processing Unit) was first time introduced in the 1980's, but it is questionable, if we can really call these historic chips as „GPU“.[18] In history, purpose of video chips was to transform bitmaps stored in the operation memory into visible images on visualization devices such as CRT (Cathod Ray Tube) monitors or later LCDs and projectors.

The simplest devices just transformed digital bitmaps into analog signal required for CRT monitors using interfaces such as VGA.

Later during the development a new idea emerged. The idea was to disburden a CPU (Central Processing Unit) by executing some very simple graphics related operations on a video chip. This concept was implemented in the next generation of video chips, which has been providing simple 2-dimensional graphical operations by itself. The first such processor was serially produced and equipped in Commodore Amiga computers in the 1980's. For better imagination, this graphic accelerator has its own simple instruction set and supported operations such as line drawing based on coordinates, filling the area with specified color or block bitmap image transfer. This was a big step ahead.
But it is still incomparable with today's GPUs.

Take a short look at the next advancement in this technology. After simple 2D graphics hardware acceleration, some requirement for unified API(Application Programming Interface) occurred. Later in 1990's also some requirements for higher resolution have appeared. But more important is that, primarily thanks to the 3D game industry, requests for 3D operations accelerated in hardware appeared. The first 3D accelerators were not very successful but it did not stop further development. As an example of this era, we can give a name of famous graphical card of this generation and it is `3dfx Voodoo`.

Later, probably for economical reasons, 2D and 3D acceleration circuits were merged into one video chip or graphical processing unit. The next generation cards finally brought compatibility to unified API. In the early 1990's the primary API was considered OpenGL and later Microsoft proprietary Direct-X.

The approach of OpenGL interface was interesting thanks to creating the pressure to move a software implementation into the hardware acceleration. In other words, new API function was created. Then implemented in software for standard CPU computation and later same API function was realized inside the graphical card hardware unburdening the central processing unit and possibly speeding up graphically oriented applications.

The newer versions of graphical API presented new acceleration possibilities of 3D transformations and lighting. This transformation and lighting hardware acceleration was later transformed into vertex and pixel shaders.

Vertex a is graphical term describing a point in 3D space defined by three coordinates (x, y, z), it also can be described as analogue of pixel(2D) in 3D space.[25]

Shaders are computer programs used to calculate rendering effects on GPU. They represent a new approach, characterized by their programmability, allowing customized effects, instead of using fixed functions or operations. This gives a direction for further development in the graphical processing units field. Shaders uses specific programming languages to define the program, for example GLSL from OpenGL or Cg from nVidia. As we see these languages are not really widely known to computer programmers or software developers.

But when shaders were standardized by OpenGL and others, new possibilities emerged for computer scientists. Using graphical cards for other than graphical computation is not such a new idea as it seams to be. Computer scientists experimented with this approach in early 1990's. Experiments prove that this could be a very promising way in the future. [9]

In the past personal computer processors (CPUs) started with frequencies around 20 MHz for Intel 80386. In these days our CPUs reach frequencies around 3 GHz which is more than 100x higher frequency. On the other hand this was not the only way used for speeding up computers. As our technology of electronics, based on CMOS (Complementary Metal-Oxide-Semiconductor) is reaching physical limits we need other ways of speeding up more and more. Our CPUs are more complex and advanced every year. Using jump prediction, several level of memory/register caching, wider words (64 bit), pipelining, ... Our CPUs overcame some slowing issues, but definitively not all.

5.1.1 Drawbacks of CPU computation

Let's point out the important drawbacks of classical CPU computational approach. CPUs are designed for serial instruction evaluation, in the past evaluating maximally one instruction per clock tick. Later advancements allowed evaluating several instruction which are not colliding and are independent of each other at the same time using different parts of the CPU. This technique is referenced as pipelining[21].

Another big drawback is that CPUs on Intel x86 architecture were primarily designed

to compute with integer values. Late in the 80386 times special integrated circuit called FPU (Floating Point Unit) co-processor was added, to allow computing with floating point numbers, using a special instruction set for this purpose. Later, in the next processor series Intel 80486 DX, FPU was integrated with CPU on one chip. And small improvements were made over time flow in those days. Next important event was addition of SSE (Streaming SIMD Extension) in Intel Pentium 3 with specific instruction set extension. This SSE addition working with 128-bit registers allowed to perform the same operation over 4 floating point numbers with single precision (32 bit) per one instruction. Or the same operation over 2 floating point with double precision (64 bit).

This approach, when the same operation is performed over 2 or more data units, is referenced as SIMD according Flynn taxonomy [17]. SIMD is an abbreviation for „Single Instruction Multiple Data“. Standard Intel x86 CPU without extensions would be classified according Flynn's taxonomy as SISD - Single Instruction Single Data, which showed in the history as an effective economical solution for computational requirements in personal computers or even in servers.

Historically SSE (earlier MMX - Multi-Media extension) for Intel, 3Dnow! for AMD brought new concept into CPU area of widely spread personal computers.

This concept was unfortunately more of a theory than practice, only very few applications made real use of 3Dnow! or SSE. This happened for several reasons, briefly said. It was because there were no standards, working on processors of both strong players, Intel and AMD. And in the most cases, using these extensions required writing assembly language using special instructions. Because automated compilers were incapable of making real efficient use of these extensions, based on optimizing source code written in programming languages designed for SISD architecture.

In comparison of these issues, possible benefit was not so high. At the best, in the ideal case, SSE could provide theoretically 4x times faster float computing, but only in the case, where the same operation over several floating point numbers is performed.

In this brief introduction to history of processing units, we cannot cover all important devices or principles. So I would like to refer you to other sources for more details. Briefly said we practically didn't mention Motorola PowerPC CPUs equipped with vector computing units, which were embedded into Apple computers. More details can be found in [22, 16].

As we saw earlier the concept is very interesting. But the area of application is not fully generic. When we say General Purpose GPU computing, somebody could understand it in the way, that everything what we are computing on a CPU, we can now compute on the GPU. This can be really possible in some cases, but definitively not effective. GPGPU is a very useful promising approach for solving higher computational requirements, where we can exploit parallelism over data processing. This can provide huge speed-ups in computationally intensive tasks. But on the other hand, using the GPGPU for tasks, which require a lot of logical control, conditional jumps and so on ... will be much less effective than by using the CPU, which is better suited for these types of tasks. This lead us to a simple conclusion that for effective computing much more is required, than just moving parts of code into the GPU to execute it. The hybrid computation, that divides tasks between the

GPU and the CPU wisely, is the way to achieve efficiency.

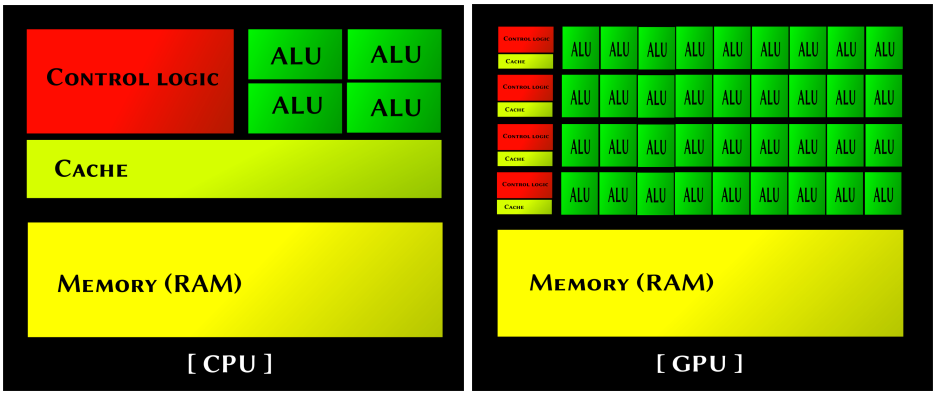


Figure 5.1: Central processing unit compared to Graphical processing unit

At this time, modern GPUs are very complex devices, equipped with special internal circuits slightly different from classical CPUs. Modern GPU chips contain inside several control logical circuits, which are more simple that their equivalents in a CPU. But as we said earlier GPUs are designed for a different purpose. Today's GPUs are still very useful in graphical operations, but their internal architecture was shifted from fixed functions, wired in the hardware, to a programmable architecture.

Inside a GPU we can now see several internal units, like stream processors, sometimes also called vector processors, which are the important parts of SIMD class of computational device. These processors perform the same operation/instruction over several input data banks (vector) and create or transform them into output data.

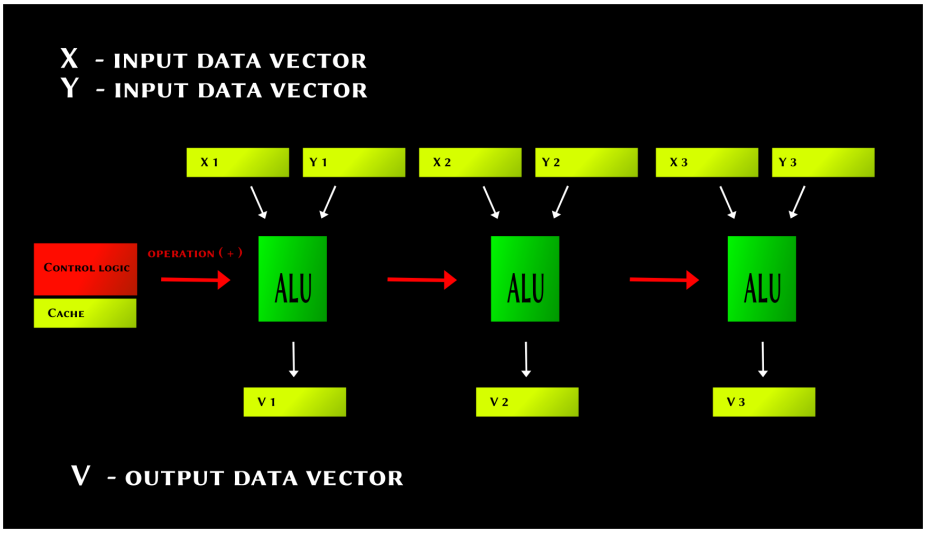


Figure 5.2: Vector processor principle

5.1.2 GPGPU related limitations

GPGPU is not an universal solution and it is not suitable for every type of algorithm. Today's GPGPU usually contains several multi-processors, where one instruction execution unit controls several ALUs.

Concept of GPGPU is based more on a slower execution of many computational threads in parallel, than fast executing one thread in the serial manner.

It implies that in situations, where is parallelization not possible, GPGPU computing cannot provide any benefit. Thanks to the typically lower processing frequency of the GPU, utilizing it in such tasks would result in slowing down, of whole computing process.

GPGPUs in these days do not have such advanced features as branch prediction, speculative execution, and etc.. These can be found in CPUs for relatively long time. So executing the code with big amount of logical branching inside is not very advised on GPU.

Heterogenous computing can be used in such cases. Heterogenous, also sometimes called „hybrid“, computing means utilizing GPGPUs in parts of code, where can be their specific computing power beneficial, and executing the rest of code on the CPU based platform.

5.1.3 Advantages of GPGPU computing

The short summary of advantages provided by GPGPU computing is briefly described here.

On the image, comparing internal architecture of a CPU and a GPU, the fact, that GPUs are massively parallel devices, can be seen. Today's GPU contain several multi-processors, which are available to perform computation in parallel, controlling bunch of ALUs by the same instruction at the time. If the program instruction code is written in parallelism allowing way, GPGPU can distribute this computational task over multi-processors and execute it at the same time.

Another one big contributions of graphical cards is also enormous computational power in floating point operations, especially in single precision (float) in the past. In these days, double precision operations are on a rise also.

Following graph visualizes rise of computational power in FLOPS(floating point operations per second) for last decade comparing CPUs and GPUs.

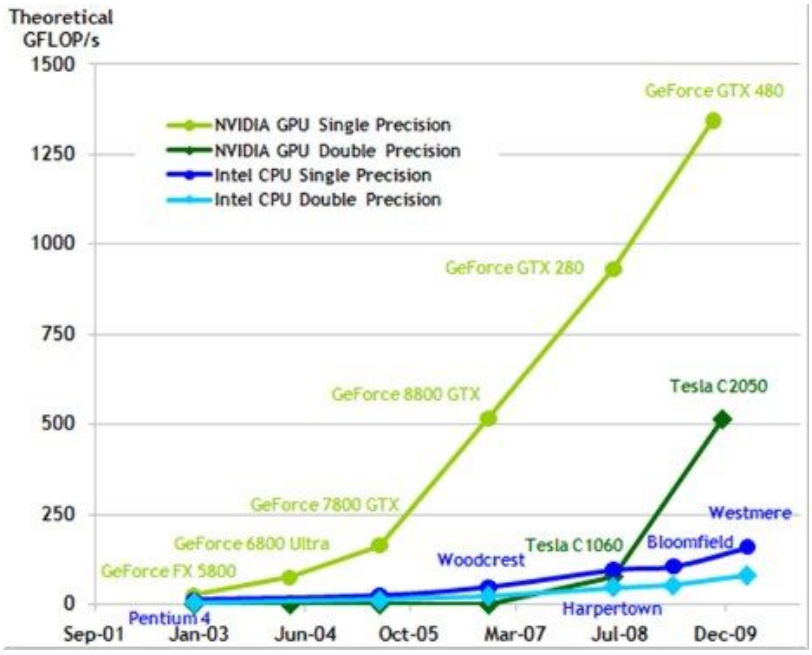


Figure 5.3: Floating point operations per second, in the past [13]

As was earlier described in this text, graphical chips made lot of steps towards todays general purpose GPU.

Today's graphical processing units have standardized general purpose programming interfaces. The most important two are CUDA and OpenCL. Both of them are described later in this chapter.

5.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by nVidia not only for a graphics processing. CUDA is proprietary technology of nVidia company. [14]
This technology provides abstract interface to GPU, through extensions in various standard programming languages. The most supported variant is **C for CUDA**. Just for sure, let's mention that CUDA works only on nVidia GPUs and that it is one of the most mature published technology for the parallel computation utilizing GPU.

The concept of this architecture is based on following abstraction elements:

- hierarchy of thread groups,
- shared memory,
- barrier synchronization.

These elements can provide possibility to divide the computational task into data parallel computing subtasks. To achieve parallel description in source code, an abstract thread concept is used. So the computational task is divided into subtasks using threads and blocks.

The decomposition does not decrease language expressive power. This is guaranteed by allowing threads to cooperate during each sub-task execution. At the same time this enables automatic scalability.

Each block of threads can be scheduled for execution on any of the available processor cores. This can be done in parallel or in serial order.

Scalability by CUDA requires, in runtime, only to know the number of physical processor cores available on the GPU to perform execution of thread blocks in the efficient way, utilizing as much cores as possible at the time.

CUDA presents several key elements, used in program definition.

- Host
- Device
- Thread
- Block of threads
- Kernel

Host is keyword, which refers to base CPU platform.

Device is keyword, which refers to GPU, used in context of execution, memory, etc.

Thread is single function execution, which is in CUDA specific 3-dimensional vector identifier .

Block of threads is organized group of certain threads amount, which can be mapped to multi-processor for execution as the smallest element.

So called **kernel** is the core of the CUDA GPU computation. Kernel is a specific C function in the C for CUDA which is executed on the GPU and divided into N threads. Each thread has it's own identifier `threadIdx` to specify the task, specific for the thread.

Typical use of the thread identifier is as an index to data array for input and output data vectors.

Lets take a look at illustration how the simple CUDA kernel can look like.

```
// C for CUDA: Kernel definition
__global__ void Vector_addition(float* X, float* Y, float* R) {

  int idx = threadIdx.x;
  R[idx] = A[idx] + B[idx];
}
```

This was brief introduction into CUDA, more implementation related details can be found in [13].

5.3 OpenCL

Open Computing Language is an open standard created by a technology consortium known as Khronos group. This standard was developed under control of the most important companies in CPU and GPU related industry. For illustration let's mention some: Intel, AMD, nVidia .... Khronos group is also responsible for OpenGL standard, [10] which gives them renown and experience in this area.

The first version of OpenCL by Khronos group was published in 2008. This standard is something slightly different than earlier mentioned nVidia CUDA. OpenCL is based on C99, but it was not designed only for the GPU computing. Abstractions provided in OpenCL allows utilization even of future devices, combining CPUs, GPUs, and other devices. But for effective execution, specific code should be created for specific execution platform.

OpenCL is open standard with bright future. At this moment it is strong competitor to nVidia C for CUDA.

Chapter 6

Algorithm, drawbacks and improvements

This part of thesis is dedicated to description of an existing algorithm implementation, it's drawbacks and also possible improvements.

6.1 Existing implementation

According BEDPOSTX homepage [6]:

„BEDPOSTX stands for Bayesian Estimation of Diffusion Parameters Obtained using Sampling Techniques. The X stands for modelling Crossing Fibres. bedpostx runs Markov Chain Monte Carlo sampling to build up distributions on diffusion parameters at each voxel.“

Bedpostx is in a fact interface to the XFIBRES, which is the core of diffusion sampling. Here will be illustrated the algorithm of XFIBRES:
(very simplified pseudo code)

```
Initialize_parameters();

Load_data();           -- 4D data
Normalize_data();      -- scaling vectors to resolution
Apply_mask_on_data();  -- to process only requested area
Transform_to_spherical_coordinates();  -- from Cartesian

for each(voxel){
  run_mcmc(voxel, data);     -- loop
}

Store_results();
```

As we can see, running MCMC(Markov Chain Monte Carlo) sampling over each voxel is the core function of BEDPOSTX. Let's begin examination of this function.

```
RUN_MCMC(){

  for( burnin_count ){
    jump();                -- single MCMC iteration

    if(adjust_counted)
      adjust_proposed_values();

  }

  for( regular_sampling_runs_count ) {
    jump();                -- single MCMC iteration

    if(adjust_counted)     -- adjusting variables after several iterations
      adjust_proposed_values();

    if(record_every_counted) -- recording only samples
      record_sample();

  }
}
```

From this simplified function pseudo-code we can see, that computing process is divided into 2 phases. The burn-in phase and regular sampling phase. Number of burn-in iterations and regular sampling iterations can be specified by user. This can be useful to adjust based on noisiness of data.

This pseudo-code description is written on high abstraction level. Internal parts of single MCMC iteration `jump()` is based on proposing several variables, based on proposals and pseudo-randomly generated numbers.

```
JUMP(){

  propose_diffusivity();           -- diffusivity in voxel

  propose_theta();                 -- phi, polar cooridnate of tensor

  propose_phi();                   -- phi, polar cooridnate of tensor

  propose_fractional_anisotropy(); -- anisotropic / isotropic diffusivity


  Evaluate_proposed_variables();
    accept / reject sample

}
```

After each proposition function is done re-computation of signal data. Where the time consumption is directly dependent on how many samples over voxel were acquired during DWI MRI scan.

It would be pointless to rewrite source code into this report, so for more details please take a look into attached source code.

6.2 Improvements

Lets take a look on what can be implemented in more efficient and parallel way.
After examining the source code, running profiler, measuring time consumption for each important part of code, the most time consuming pieces of code, have been identified. Lets name some of them:

- compute_signal()
- compute_iso_signal()
- compute_likelihood()

Each of these functions contains one or more loops. These loops are computing data for each performed MR scan where angle, intensity or both differs. When we have this data recorded from previous phases of the software preprocessing, the modeling the diffusivity inside the voxel is performed. When we visualize the real recorded data, we would see something similar to the following image.

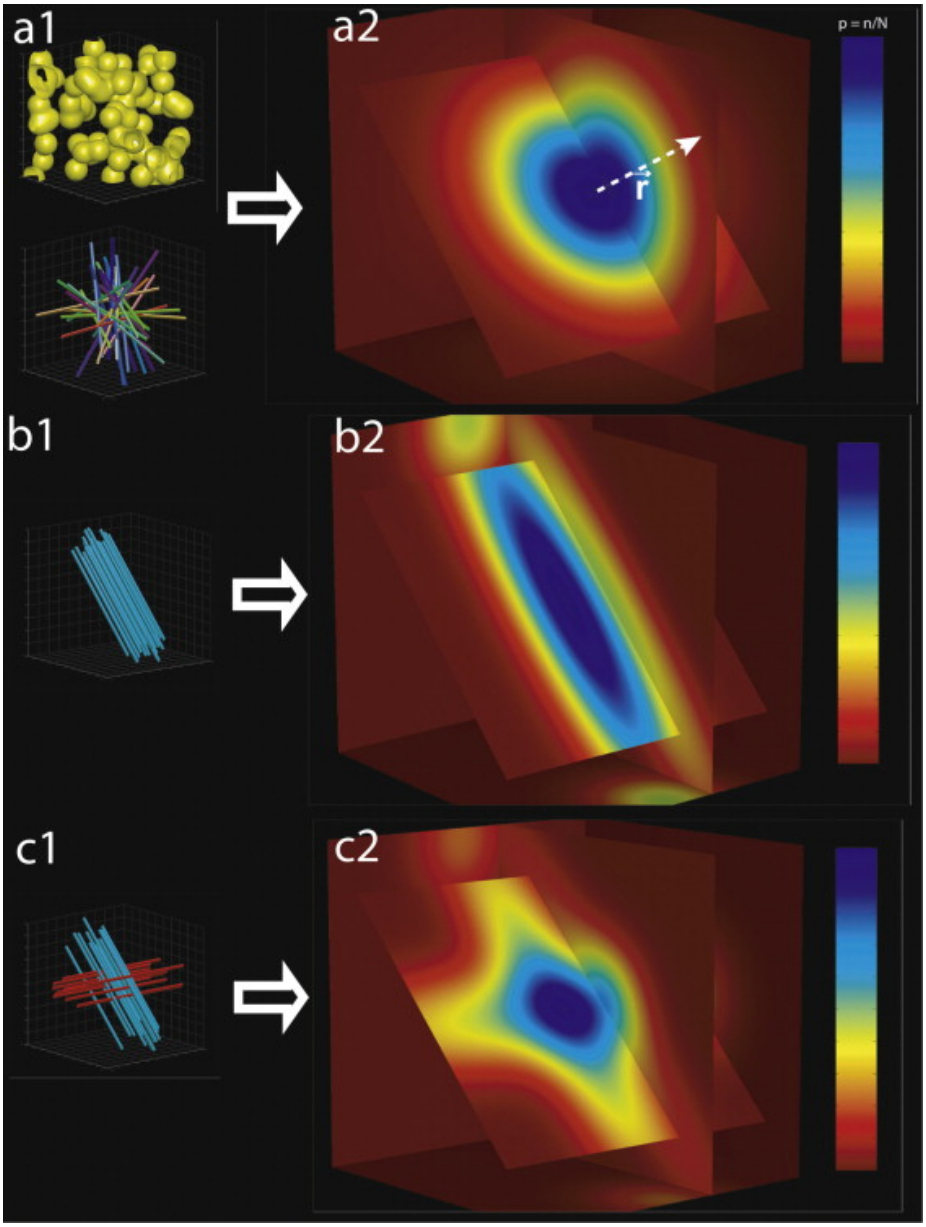


Figure 6.1: Voxel diffusion variants [11]

6.2.1 Variant 1

User of the software can specify how many fibers wants to model within the voxel. What model of diffusion should be use, etc.. XFIBRES has many options which can allow us to model diffusion shape inside a voxel.

This can be very beneficial for neuro research, but on the other hand it provides complicated source code, with the higher possibility of a slow execution.

After several experiments and analyzes decision to write above mentioned functions in C for CUDA extension has been made, to exploit possibility of running parallel computation

within these functions instead of serial loops execution. This was possible because these parts of code operated on vector data, the vector of measured signal, the vector of isotropic signal, vectors of measuring angles and etc..

After new measuring it showed to be, that execution on GPU is faster in some cases, but also much slower in others. There exist several reasons to explain this behavior.

When communication between the CPU and the GPU is performed, latency effect should be taken in consideration. The bandwidth of data transfer is huge enough to cover latency (delay) when the larger amounts of data are transfered.
But thanks to the Markov chain effect these transfers need be performed in the each iteration on the small amount of data.

Another reason for slowing down is the kernel invocation delay, which can be by my measurements up to 10 microseconds per invocation. When the kernel invocation costs similar time as the whole execution of the kernel, this effect can break down the acceleration, achieved by parallel code execution.

The last important slowdown is produced by synchronization of threads. If the synchronization of threads is needed, this brings another delay, for the example when we need to summarize the vector of values in the parallel manner we need to used shared memory (for threads), and the proper access to shared memory requires synchronization. Data transfers from CPU to GPU also can require certain type of synchronization.

This have shown to us, that serial execution can be sometimes faster, than parallel one, thanks to the overhead.

In order to achieve acceleration, I made more changes, than just moving some parts of code execution to the GPU. To achieve faster execution abstract vector data structures have been transformed into simple data arrays. Here was encountered first inconsistency of the computed results. After the investigation, I found that all numeric expressions are computed using single (float) precision but `ColumnVectors`, are stored in the double precision floating point format.

Further investigation showed that even final results of the program are stored in the single floating point precision. That led me into conclusion that using double inside `ColumnVector` objects have been done by mistake of developers because, the precision of these data objects depend on the way, how is the `NEWMAT` library linked, and which compiler macros are defined.

As we can see from the graph comparison of single and double precision floating point computing power, the single precision operations are much wider supported, allowing faster serial computing on 32-bit architectures or more values computed at same time on the GPU, or the SSE. After considering this effect, single floating point number precision was chosen. The paradox here is, that it appeared to produce some slowdown on 64-bit CPU architecture in dependency of which optimization parameters of GCC compiler are used.

Branching inside each loop iteration, to determine which diffusion model to use, or how

many fibers to model, etc. I considered as very ineffective implementation. So in the cases where it appeared to be reasonable, more code have been expanded into very similar functions to eliminate this almost redundant evaluating and branching.

Loop constructions to copy values between the vectors, have been replaced by effective calls of `memcpy()` function over simple data arrays, computing as less as possible, to achieve the higher efficiency.

6.2.2 Variant 2

Using hybrid computing which runs the serial part of code on CPU and vector operations on GPU did not show to bring a huge acceleration. Based on that different approach has been chosen.

This different approach exploits the possibility of running MCMC on each voxel independently in parallel. This means executing almost whole code on GPU. That brings the necessity to implement originally object-oriented C++ code to low level C for CUDA. This is needed for execution as `CUDA kernel`.
To achieve this goal, Multifibre object data representation was transformed into the one dimensional array, where position of each variable or vector of variables has been marked by static offset. This means also effective memory transfers, where just 1 block of memory is copied by memcpy instead of slow copy loops. For effective execution of computation in parallel is then just needed to set pointer to the specific voxel variable array inside the CUDA kernel.

Time requirements are enormous primarily for `run_mcmc()` function, which is executed in the original implementation sequentially for each voxel.
With knowledge of time requirements for each important functions, decision was made to concentrate acceleration on the most time consuming functions. This means that functions, which are executed once per voxel or once per run of the whole application and do not consume lot of time, are computed in serial manner.
Another idea, how to speed-up computing, have been applied. Required pseudo-random generators for uniform and normal distribution run on the CPU before the first voxel is initialized. Two arrays of pseudo-random number are pre-generated and then loaded into the GPU memory. Each voxel has it's own index into random number field, which is just incremented when pseudo-random number is requested. This feature brings a certain time saving and also more determinism when each voxel uses the same sequences of random numbers.
The whole main computational process is divided into bursts, where the defined number of voxels is initialized sequentially, then transfered into GPU memory.

```
for( all voxels in burst )
  Initialize_voxel(voxel);
  Copy_voxel_into_GPU(voxel);
}

run_MCMC(burst_first_voxel, VOXELS_AT_ONCE, ...);

for( all voxels in burst )
  Retrive_voxel_samples(voxel);
  Process_recorded_samples(voxel);
}
```

Measured speed-up and time consumption of this approach is discussed in the next chapter.

Chapter 7

Experiments and testing

7.1 Reliability of results

In the medical application of software is reliability of diagnosis method extremely important. This chapter describes how experiments were performed and compared to original (non-accelerated) implementation.

7.1.1 Comparing output results

Software, which works with fully deterministic methods have to provide the same results each time it runs with the same input data. In the situation, where is used some „random“ element, is situation more complicated. To achieve fully deterministic and repeatable execution we have to use pseudo-random number generators instead of truly random ones. Pseudo-random generator require the so called „seed“ to initialize the process of generating numbers. If the same „seed“ is used to initialize the same random generator, each run generate the same sequence of numbers.

As we earlier pointed, the different precision of floating point operations made binary comparison of the output data impossible.

Later during the experiments even possible bug was reveled in original implementation, when the simple move of precision to the higher level, results differed in more than 20%. In some voxels the change was of dominant tensor angle very small, as was expected. But in few voxels, the change of angle have been almost orthogonal to original direction. This can be considered as very disturbing. But my further investigation led me to conclusion, that this can happen by modeling more than one fiber crossing the voxel, in such cases can be primary and secondary diffusion tensor swapped.

These effects of computation forced me into manual examination after each single change of source code.

To make some development even possible, the small subset of data have been created to perform these experiments in a reasonable time.

For the testing has been used the slice of the human brain, containing 240 voxels.

7.2 Measuring computational time

This section provides information primary about time consumption of computation. Also information about specific performed experiments is provided. Several experiments were performed, using different process parameters.

...

```
---------------------------------------------------------------
Hardware description of computing device:

  CPU: AMD Phenom(tm) II X4 965 Processor
  cache size: 512 KB
  used 1 core

  GPU: nVidia Corporation GT200 [GeForce GTX 260]
  CUDA Driver Version:                           3.20
  CUDA Runtime Version:                          3.10
  CUDA Capability number:                        1.3
  Clock rate:                                    1.24 GHz


---------------------------------------------------------------
Software:
  Linux HIVE 2.6.32-25-generic x86_64 SMP

XFIBRES have been executed with following parameters:
  voxels = 240

  nfibres = 2
  fudge = 1
  burnin = 1000
  njumps = 1250
  sampleevery = 25
  model = 1
  bvals count performed by MRI scan = 129

time [s] - measured by process execution time
---------------------------------------------------------------
ONLY DATA INITILIZATION                          TIME:   4.29 s
REFERENTIAL CPU implementation of XFIBRES(4.1.8)  TIME: 189.52 s
 0.77s per voxel
```

```
-----------------------------------------------------------------
VARIANT 1
-----------------------------------------------------------------
xfibres - compute_signal on GPU                   TIME: 114.53 s
 0.45s per voxel
xfibres - compute_signal, compute_iso_signal      TIME: 120.44 s
xfibres - all vector functions on GPU             TIME: 322.76 s
-----------------------------------------------------------------

-----------------------------------------------------------------
VARIANT 2
-----------------------------------------------------------------
CPU pseudo-random generating: 65536 numbers
  time:    0.0114288 s

VOXELS_AT_ONCE / KERNEL_BLOCKS 64                  TIME:  5.32 s
  kernel call iteration: 0.221459 s

VOXELS_AT_ONCE / KERNEL_BLOCKS 32                  TIME:  6.23 s
  kernel call iteration: 0.218717 s

VOXELS_AT_ONCE / KERNEL_BLOCKS  1                  TIME: 55.55 s
  kernel call iteration: 0.215441
-----------------------------------------------------------------
```

From these measurements we can compute the speed up.

```
  (189.52 - 4.29)  /  (114.53 - 4.29) = 1.68 x
  (0.77)  /  (0.45) = 1.70 x
```

This means that GPU accelerated implementation, variant 1 is `circa 1.7x faster` than the original referential CPU-only implementation on this specific hardware configuration in these parameter settings.

```
  (189.52 - 4.29)  /  ( 5.32 - 4.29) = 179.84 x
```

But more interesting are results of GPU accelerated implementation variant 2. Which is `circa 180x faster` than the original referential CPU-only implementation on this specific hardware configuration in these parameter settings.

Chapter 8

Conclusion and future work

As was expected, the parallel implementation based on principles of general purpose GPU computing seems to be more efficient, than the original CPU-only implementation. On the other hand, certain parts of code were more efficient, when the execution was performed on the conventional CPU.

Several parts of computing algorithms were parallelized, especially the parts which consumed the biggest percentage of computing time and where the parallelism had some potential to bring some speed up.

Utilization of GPUs in this specific area, for computing the neural diffusion tensor „map“, showed to be efficient for the acceleration. It also pointed at the possible GPU's utilization in others computation intensive MRI-related software implementations can be very beneficial. Especially in the computation with floating point numbers, matrices and vectors.

Certain parts of remaining code still use serial CPU execution, because as was earlier explained GPU computing is not ultimate solution for all types of computational tasks and transformation of certain CPU code into GPU would provide only very small computational time benefit.

The acceleration is a never ending process, so every time a code is analyzed, new possibilities of speed up occurs. But the important question is, how big benefit can each change of the code provide. How close to specific hardware is wise to go, ...

I would like to point at the future work, the possibility to transform the whole per-voxel independent code into independent blocks utilizing only GPGPU computing to perform whole computing task have been made. This provided huge unburdening the CPU, and it would allow on the modern multitasking operating system to run also CPU implementation at the same time. This will require to do complete re-design of certain code parts, and to perform that, the longterm cooperation with FMRIB research team will be needed. This software is now starting it's testing period to become official part of FMRIB FSL tool-set. For later is also scheduled a software development process for a fully automatic results comparator to ensure correctness of the results.

Bibliography

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. *An Introduction to MCMC for Machine Learning.* Machine Learning 50(1), 2003.

[2] P. J. Basser, J. Mattiello, and D. Le Bihan. *Estimation of the Effective Self-Diffusion Tensor from the NMR Spin Echo.* Journal of Magnetic Resonance Series b 103(3), 1994.

[3] C. Beaulieu. *The basis of anisotropic water diffusion in the nervous system - a technical review.* NMR in Biomedicine 15, 2002.

[4] D. Le Bihan, E. Breton, D. Lallemand, P. Grenier, E. Cabanis, and M. Laval-Jeantet. *MR imaging of intravoxel incoherent motions: application to diffusion and perfusion in neurologic disorders.* Radiology 161(2), 1986.

[5] FMRIB. Characterization and propegation of uncertainty in diffusion weighted mr images. http://www.fmrib.ox.ac.uk/analysis/techrep/tr03tb1/tr03tb1/, 2004. [Online; accessed 01-July-2012].

[6] FMRIB. Fdt - bestpostx. http://www.fmrib.ox.ac.uk/fsl/fdt/fdt_bestpostx.html, 2012. [Online; accessed 01-July-2012].

[7] FMRIB. Fdt processing pipeline. http://www.fmrib.ox.ac.uk/fsl/fdt/fdt_pipeline.html, 2012. [Online; accessed 01-July-2012].

[8] FMRIB. Global connectivity estimation. http://www.fmrib.ox.ac.uk/analysis/techrep/tr03tb1/tr03tb1/node14.html, 2012. [Online; accessed 01-July-2012].

[9] GPGPU.org. History of gpgpu. http://gpgpu.org/oldsite/data/history.shtml, 2012. [Online; accessed 01-July-2012].

[10] Khronos group. Opencl specification. http://www.khronos.org/opencl/, 2012. [Online; accessed 01-July-2012].

[11] Patric Hagmanna, Leila Cammounb, Xavier Gigandetb, Stephan Gerhardb, P. Ellen Grantc, Van Wedeend, Reto Meulia, Jean-Philippe Thiranb, Christopher J. Honeye, and Olaf Sporns. Mr connectomics: Principles and challenges. http://www.sciencedirect.com/science/article/pii/S0165027010000361, 2010. [Online; accessed 01-July-2012].

[12] Johansen-Berg, Behrens, and T.E.J., editors. *Diffusion MRI: from quantitative measurement to in vivo neuroanatomy.* Elsevier Science, London, 2009.

[13] nVidia. Nvidia cuda c programming guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 2010. [Online; accessed 01-July-2012].

[14] nVidia. Cuda. http://www.nvidia.com/object/cuda_home_new.html, 2012. [Online; accessed 01-July-2012].

[15] S. M. Smith, M. Jenkinson, C. F. Beckmann M. W. Woolrich, T. E. Behrens, H. Johansen-Berg, P. R. Bannister, M. De Luca, I. Drobnjak, D. E. Flitney, R. K. Niazy, J. Saunders, J. Vickers, Y. Zhang, N. De Stefano, J. M. Brady, and P. M. Matthews. *Advances in functional and structural MR image analysis and implementation as FSL.* NeuroImage 23, 2004.

[16] Wikipedia. Cpu — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Central_processing_unit, 2012. [Online; accessed 01-July-2012].

[17] Wikipedia. Flynn's taxonomy — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Flynn's_taxonomy, 2012. [Online; accessed 01-July-2012].

[18] Wikipedia. Graphical processing unit — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Graphics_processing_unit, 2012. [Online; accessed 01-July-2012].

[19] Wikipedia. Neuron — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Neuron, 2012. [Online; accessed 01-July-2012].

[20] Wikipedia. Physics of magnetic resonance imaging. http://en.wikipedia.org/wiki/Physics_of_Magnetic_Resonance_Imaging, 2012.

[21] Wikipedia. Pipeline (computing) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Pipeline_(computing), 2012. [Online; accessed 01-July-2012].

[22] Wikipedia. Powerpc — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/PowerPC, 2012. [Online; accessed 01-July-2012].

[23] Wikipedia. State of the art. http://en.wikipedia.org/wiki/State_of_the_art, 2012.

[24] Wikipedia. Tractography. http://en.wikipedia.org/wiki/Tractography, 2012.

[25] Wikipedia. Vertex — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Vertex, 2012. [Online; accessed 01-July-2012].

[26] Wikipedia. Voxel — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Voxel, 2012. [Online; accessed 01-July-2012].